University of Tsukuba
Department of Computer Sciences

Urban Settlement Generator

Report

Authors :
Eduardo HAUCK
Adrien CÉLÉRIER
Théo LOUBET

Supervisor :
Claus ARANHA

Introduction

The Urban Settlement Generator is an Artifical Intelligence that aims at generating a modern settlement on a given unknown Minecraft map.

To perform such a task, the framework MCedit, a program developed primarily by *codewarrior0* to enable Minecraft players to freely edit their Minecraft worlds outside of the game, is the ideal tool. It is therefore possible to code the AI by using this program, manipulating a Minecraft map file using a customized python script, called a *filter*.

The settlement is composed of different constructions, that are generated according to a downtown/outskirts division, with apartment buildings placed towards the center and other buildings such as houses or farms placed towards the edges of the map. All these buildings are then connected by roads and bridges to cross areas filled with water.

This project is the continuation of the entrie presented at the 2019 edition of the GDMC *(Generative Design in Minecraft Competition)* by *ehauckdo*.

Building Lots and Construction Process

The first step, after receiving a portion of the map selected by the user in *MCEdit* in the perform function, is to take this space and go through an initialization process, that consists in different steps, like generating a height map and a 2D matrix containing the height (y coordinate) of the first valid ground block for all coordinates x, z of the map.
Therefore, all the trees in the map are removed so that those that are not intersecting with any settlement can be replaced once its generation is complete. The whole space if also divided into a city center, composed of a percentage of the central area of the whole space, while the remaining area is divided into 4 sections that become the neighbourhood area.

The center and the neighbourhood go through binary or quadtree (chosen randomly each time) space partitioning 100 times and the generated partitions are added to a list. Each partition becomes a building lot, and go through different tests to remove the lots that are invalid. Checking the size, steepness and the nature of the blocks that composed the lot, the lots that are either too small, too steep or that have water/lava blocks are considered invalid. The list is then ordered according to the steepness of the lots.
The steepness score of a building lot is calculated by summing the absolute difference between the mode of the heights of the lot, and the height of every other block of the lot. Therefore, the higher the steepness score, the steeper the lot is, a score of 0 meaning a building lot is a flat surface.

All the intersecting lots are then removed from the list. If the remaining valid lots are not enough to build a minimum amount of constructions, the process of space partitioning is repeated with a higher minimum steepness score until it gets enough buildable lots.

To generate a structure, the first thing to be done is to check whether the lot terrain is flat. If not, earthworks are carried out to ensure that the perimeter is at the same height level. This is done by finding the most occurred height, and then flattening the entire lot to match that. The most occured block around the lot is used as the base block for flattening the area.

In the city center, only one type of construction is built on the lots, called **building**. In the neighboorhood area, the building type differ based on the steepness of the building lot, **houses** are built on the 50% flattest lot, **farms** on the next 20% lots. On the 30% remaining lots that are assumed very steep, attemps to generate **roller coasters** are proceeded, and **towers** are generated instead in case of failure.

Structures

The generation of structures is mostly hardcoded. For most of the structures, it goes as followed :
First, the land is cleared changing unnecessary blocks (trees...) to air blocks. Then a random width and depth between a certain range are set. From that, the main part (floor, walls...) is generated, an orientation (N/S/E/W) is decided based on the position of the structure, always facing the center of the map. The orientation will dictate the location of the entrance door and the entrance to the lot at the edge of it where roads will connect.
Some other structures can however have totally different process of generations (roller coaster...).

Here is a brief description of all sorts of structures in the settlement :

House

Houses have walls made of double stone slab and pitched roofs made of wood. They have one oak door as the entrance on one side, and glass windows on the walls perpendicular to the door wall. Houses are furnished with carpets, a table in the center of the room, a chandelier on the ceiling, and a bed, bookshelf and couch in the corners.

Building

Apartment buildings have clay walls, 4 to 10 floors, with one apartment on each floor. The entrance to each building leads to a small corridor with a door to the ground floor apartment and a stairwell for reaching other floors. The apartments' furniture follows a similar fashion as in the houses, and windows are on the opposite wall to the building entrance.

Farm

Farms are surrounded by oak fences to separate them from the rest of the city. A grassy path starting from the entrance, an oak fence gate, connects the structure to the roads.
Inside the farm, cultures (carrots, potatoes...) are randomly generated following differents patterns ; the main one being lines of water surrounding the farm's plantations, or others like a "smiley farm" that is drawing a smily face watchable from the sky.

Roller coaster

Roller coasters are a system of rails looping to create a structure on a slope without modifying the terrain. It therefores follows a different pattern of construction compared to the other structures.
The algorithm consists of finding, from one of the highest points of the lot, the longest path that fulfils the Minecraft rails requirements (a rail that goes down cannot turn...), going down and staying at the same level for a maximum length of 5 blocks (to avoid snake-shape rails especially at the end of the path).
It therefore tries to connect the starting point to the ending point going on another way than the initial one so that the whole system loops.
Every "going-down" rail is a rail powered by a redstone torch 2 blocks beneath it ; while all the others are regular rails. Beneath each rail is placed an oak wood log, and a chest is placed close to the starting point.
In case the aglorithm fails to find a path that fulfils all these requirements, or if the chosen path is too short, a Tower is generated instead.

Tower

Tower, along with the Roller Coaster, is a type of structure that does not require to flatten the lot before the construction. It is a building relatively narrow but high so that it can be built on very steep terrain. The tower is made out of bricks, inside are findable a chest and furnace at the bottom, and a wooden staircase that goes to the top of the tower, where there are windows, and a little platform with a crafting table and an anvil.

Connecting Building Lots

Once all the buildings have been placed, the last step consists of connecting them with roads and bridges, which will be respectively in dirt and wood if the road leads to a farm, or otherwise in stone. To do this, each lot is considered as a node in a graph, and a minimum spanning tree (MST) is created to connect them. Distances between nodes are calculated with Manhattan distance. With the minimum spanning tree now created, the next step is to find if some bridges are needed in order to cross areas filled with liquid.
To do so, for each edges of the MST, a simple A* algorithm runs with no consideration of height for the cost between each node, and therefore returns the simplest path possible. Then the material of each block that compose this path is checked, and if some are water or lava blocks, a bridge will be built to connect the locations where the path enters a liquid, and when it goes out of it.

A bridge is made out of slabs, which means it can gain half a block of height for every block of length, so depending on the height of the extremities and the length of the bridge, it can be built differently. If the height of the extremities are close, the bridge goes up from both side and is built in the shape of an arch. If the length of the bridge is not enough to build it that way, it tries to reach the highest endpoint among the extremities by going up from only one side. If the bridge is still unbuildable, its construction is cancelled (e.g. it can happen when a bridge is trying to connect a cliff and a beach).
If a bridge is successfully built, its endpoints are qualified as intermediate nodes, that the path must reach if there are any between two nodes of the MST.

After building needed bridges, A* is used again to actually find the path to generate the road. The heuristic function that gives the cost between the current and the target point is the Manhattan distance, and this time the cost function between each node is $1+n^2$, where n is the difference of height between the current block and the next. This function is employed in order to try and find a path that is not too steep between two points (e.g. going around a mountain instead of climbing it).
A* returns a sequence of coordinates 1 block wide where the pavement is to be generated. The road is then completed on both sides of this path to make it 3 blocks wide if nothing prevents it (water blocks, rails...).

When the pavement is finished, we go through the path another time to put ladders where the height difference between two blocks is higher than 1 and we enlighten the path. To have interesting places to put the lamps, we compute the center of mass of each portion of twenty block of the path, and put a lamp in this area if this point or one of its neighbors corresponds to a valid block, otherwise, the lamp is placed directly on the path.

Tree Gestion

Since *Minecraft* maps are almost always dotted with trees, building things while not caring about them would end up in awkward cut trees that would not be esthically appealing. Therefore, a tool had to be created to take care of this problem.

Before putting anything new on the map, every grounded block is checked to find all the trees on the map and save them. The first trunk block of every tree is saved as origin points. Then the algorithm goes up level by level until it reaches the top of the tree, and for each new height, it finds all the leaves that are at a reasonable distance from the trunk by spreading out around the trunk and save all these blocks in a data structure. Knowing that most tree foliage is touching each other, it is not possible to just spread out until it cannot find any more leaves since it could save excess leaves that do not belong to the original tree, so it has to stop seeking after a certain distance from the trunk at the risk of not saving some leaves.

After every tree is saved, all tree origins are used as starting nodes for a Breadth First Search algorithm to find all the tree blocks (even the leaves that were not saved as belonging to a tree) and erase them.

When the generation of the settlement is finished, it goes through every tree saved, check their validity (e.g. not intersecting a structure, not above a path etc.) and put back all those that are valid.

Time Consumption

Since the settlement needs to be generated in less than 10 minutes, it could be interesting to pay attention to how long each part of the generation takes. After trying to generate 3 different settlements on each map of the previous GDMC competition, the whole process took between 4 and 6 minutes depending on the map, with around 40% of this time dedicated to the initialization part of the generation (e.g. generating the height map, saving trees, etc.) and more than 40% of the total time is taken by the space partitionning of the map to find enough valid building lots. Then about 5% is concerning the building of the lots and bridges, finding and building the path. The rest of the time is taken by actually updating the *Minecraft* map.

Some potential upgrades were possible for the steepness score calculation function but it has been restricted in the final version of the project. At first, it looks interesting to be able to reduce the impact on the score of different heights compared to the mode according to the number of appearances of this height, but in the end it appeared that this idea should be abandoned because this function has to be called hundreds of times, so the qualitative gain is not profitable compared to all the time that is lost.

The total generation time is far from exceeding the time limit for the competition, and most of the time used is for generating big data structure that are then constantly reused like the height map, so there is still plenty of room to add things and improve the generator.

Potential future improvements

Several improvements are possible and affordable for the generator.
Firstly, it could be a good idea to get rid of generating the buildings on lots that need to be flatten, and therefore do the same thing that is done with the towers for every building. Doing that would saves some space and give an overall better looking map without those big flat areas.
Another improvement that could be great would be to go further in separating the city center and the neighbourhood, foe example handling them separatly, like having independant paths for each neighbourhoods and for the city center, and main roads to connect them. Each neighbourhood could also have a distinct architecture style depending on the biome it is in.